SENG 438 - Software Testing, Reliability, and Quality

Lab. Report #2 – Requirements-Based Test Generation

Group #:	**19**
Student Names:	Eric Renno
	Ryan Sommerville
	Quinn Ledingham
	Kaumil Patel

1 INTRODUCTION

The purpose of this lab is to explore how to use Junit tests to thoroughly test the methods of a particular class in order to ensure that the class functions as it should. To do this, the Junit class allows you to create a test function and compares the correct result with the result that the function actually returns. Although the process is straightforward, accurately testing a class requires a thorough analysis of boundary cases and other necessary test cases for each method. Additionally,

2 DETAILED DESCRIPTION OF UNIT TEST STRATEGY

To ensure that the JFreeChart application could be reliably tested, a test plan was constructed. This was done in the form of a .docx file that was shared over google drive. After spending time reviewing the JFreeChart documentation, test methods were chosen from both the Data Utilities class and the Range Type class. Some of the methods chosen contained int and double data types, leaving a large range of potential input values to account for. To resolve this issue, the input set was divided into the equivalence classes shown below.

Input Partition Sets for int data type (A_n):

Legal values

$$A_1 = 0 \leq x \leq 2^{31} - 1$$

$$A_2 = -2^{31} \leq x < 0$$

Illegal values

$$A_3 = -2^{31} > x$$

$$A_4 = x > 2^{31} - 1$$

$$A = A_1 \cup A_2 \cup A_3 \cup A_4 \text{, where } a_i \in A_i$$

Input partition sets for double data type (B_n):

Legal values:

$$B_1 = 1.79769313486231570 \times 10^{308} \geq x \geq -1.79769313486231570 \times 10^{308}$$

$$B_2(Min\ with\ x > 0) = 4.9 \times 10^{-324}$$

$$B_3(Min\ with\ x < 0) = -4.9 \times 10^{-324}$$

Illegal values:

$$B_4 = 1.79769313486231570 \times 10^{308} < x$$

$$B_5 = -1.79769313486231570 \times 10^{308} > x$$

$$B = B_1 \cup B_2 \cup B_3 \cup B_4 \cup B_5\ , where\ b_i \epsilon B_i$$

Equivalence class A contains all permissible and non-permissible integer arguments while class B contains all the permissible and non-permissible double values that can be used as function arguments. One important note to add is the assumption that was made in the derivation of class B, specifically partition B1. Because java double-precision follows IEEE754 values and operations, the entire set of input values is very large. Therefore, it is assumed that the values contained within partition B1 have low precision. In order to address this concern, partition B2 and B3 were abstracted.

For this assignment, only partition A1 was used for testing the integer input values. This decision was made for two reasons. The first reason is that the methods under test only used integer values for specifying rows and columns within a table (referring to methods calculateColumnTotal and calculateRowTotal) which cannot be negative. The second reason is because partition sets A3 and A4 contain values outside the acceptable int data type range which will clearly cause a runtime error. For equivalence class B, only partitions B1, B2, and B3 were considered in the test design process. The reasoning behind this is that partitions B4 and B5 only contain values outside the range of double datatype, which will cause a runtime error.

To create test cases, boundary cases were considered. These included arrays with no values, or only one row or column, or many. For values, both positive and negative values were attempted.

Following the derivation of the equivalence classes and the boundary cases, additional test cases were created to handle the inputs of the more complex data types. The extra test cases included: uninitialized arrays, empty datasets, and null values.

The test was performed using the Eclipse IDE. First, java files were created for the test methods using a shortcut on Eclipse. This will create a method in the test classes for each method to be tested in the classes to be tested. Next, test cases were invented for each method so that boundary cases are covered. Once this is done, the test cases are coded. If necessary, a Mocker class is used as a substitute for inputs where we don't have access to the actual class. After all the test cases are coded, the tests are run and the results recorded.

The benefit of using mocking is that it limits the dependencies of a test. It allows for you to test a class independently even when it depends on another class. This makes the scope of the test smaller. A drawback is that the tests that use mocking are made for a specific implementation of

the methods. This might cause you to have to also change the tests when you change the code. If you do not, it might keep passing tests that it should not be.

3 TEST CASES DEVELOPED

Tested with version 1.0. Version 1.0.19 fixes all these problems.

Class	Method	Test Function	Expected Result/ Actual Result	Passes
DataUtilities	calculateColumnTotal(Values2D data, int column)	calculateColumnTotalForTwoValues() Inputs: Mock Values2D object with 1.25 and 2.5. Partition: B1	3.75	Pass
		calculateColumnTotalForNegativeValues() Inputs: Mock Values2D object with 1.25, 2.5 and -5.5. Partition: B1	-1.75	Pass
		calculateColumnTotalForOneValues() Inputs: Mock Values2D object with two columns, the second with 1.25. Partition: B1	1.25	Pass
		calculateColumnTotalForZeroValues() Inputs: Mock Values2D object with no values. Partition: n/a	0	Pass
	clone (double [][] source)	cloneFor1By1ArrayTest() Inputs: {{10.5}} Partition: B1	Output is a clone of the input and is not a reference to the input.	Fail
		cloneFor1By10ArrayTest() Inputs: Random 1 x 10 array. Partition: B1	Output is a clone of the input and is not a reference to the input.	Fail

		cloneFor10By1ArrayTest() Input: 10 by 1 Array with random numbers. Partition: B1	Output is a clone of the input and is not a reference to the input.	Fail
		cloneFor10By10ArrayTest() Input: 10 by 10 array with random numbers Partition: B1	Output is a clone of the input and is not a reference to the input.	Fail
		notCloneFor10By10ArrayTest() Input: 10 by 10 Array Partition: B1	Output is a clone of input, and doesn't equal a different array.	Fail
	createNumberArray(double[] data)	createNumberArrayForNull() Inputs: NULL Partition: n/a	Exception	Pass
		createNumberArrayOfSize1() Inputs: double [] = {10.5} Partition: B1	Number [], {10.5}	Fail
		createNumberArrayOfSize10() Inputs: double[10] with random values Partition: B1	Number [10] with random values	Fail
	createNumberArray2D(double[][] data)	createNumberArray2DForNULL( ) Input: Null Partition: n/a	Exception	Pass
		createNumberArray2DFor1By1Array() Input: double [][] with 1 value Partition: B1	Number [][] array with the same value	Fail
		createNumberArray2DFor1By10Array() Input: double[][] that is 1 by 10 with random values	Number [][] array with the same length and values.	Fail

		Partition: B1		
		createNumberArray2DFor10By1Array() Input: double[][] that is 10 by 1 with random values. Partition: B1	Number [][] array with the same length and values.	Fail
		createNumberArray2DFor10By10Array() Input: double[][] that is 10 by 10 with random values. Partition: B1	Number [][] array with the same length and values.	Fail
	equal(double[][] a, double[][] b)	equalFor1By1ArrayTest() Input: Two 1 by 1 arrays with identical values. Partition: B1	True	Fail
		equalFor1By10ArrayTest() Input: Two 1 by 10 arrays with identical values. Partition: B1	True	Fail
		equalFor10By1ArrayTest() Input: Two 10 by 1 arrays with identical values.	True	Fail
		equalFor10By10ArrayTest() Input: Two 10 by 10 arrays with identical values. Partition: B1	True	Fail
		notEqualFor10By10ArrayTest() Input: Two 10 by 10 arrays with different values. Partition: B1	False	Fail
	getCumulativePercentages(KeyedValues data)	getCumulativePercentagesForThreeValues() Inputs: Mock KeyedValues object with values 2, 1, and 2.	New values should be 0.4, 0.6, 1.0	Fail

		Partition: B1		
		getCumulativePercentagesForZeroValues() Inputs: Mock KeyedValues object with no values. Partition: n/a	Empty KeyedValues object	Pass
Range	combine(Range range1, Range range2)	combineTestIntersect() Input: range1: 0 to 10 range2: 5 to 15. Partition: B1	range.lower = 0 range.upper = 15	Fail
		combineTestNoOverlap() Input: range1: 0 to 10 range2: 15 to 20 Partition: B1	range.lower = 0 range.upper = 20	Fail
		combineTestNull() Input: range1 = NULL range2: 0 to 10 Partition: B1	range.lower = 0 range.upper = 10	Pass
	getLowerBound()	getLowerBoundTest() Input: Range = -10 to 10 Partition: B1	-10	Pass
	getUpperBound()	getUpperBoundTest() Input: Range = -10 to 10 Partition: B1	10	Fail
	constrain(double value)	constrainTestMiddleOfRange() Input: value = 2.5, Range = -2 to 7 Partition: B1	2.5	Pass
		constrainTestOutsideRangeAbove() Input: value = 8, Range = -2 to 7 Partition: A1 and A2	7	Pass

		constrainTestOutsideRangeBelow() Input: value = -3, Range = -2 to 7 Partition: A1 and A2	-2	Fail
		constrainTestOnLower() Input: value = -2, Range = -2 to 7 Partition: A1 and A2	-2	Pass
		constrainTestOnUpper() Input: value = 7, Range = -2 to 7 Partition: A1 and A2	7	Pass
		constrainTestOnMin() Input: range1: 0 to Double.MIN_VALUE Partition: B1	Double.MIN_VALUE	Pass
		constrainTestOnMax() Input: range1 0 to Double.MAX_VALUE Partition: B1	Double.MAX_VALUE	Pass
	getCentralValue()	centralValueShouldBeNegative() Input: Range: -10 to 5 Partition: B1	-2.5	Pass
		centralValueShouldBeZero() Input: Range: -10 to 10 Partition: A1 and A2	0	Pass
		centralValueShouldBePositive() Input: Range: -5 to 10 Partition: A1, A2, and B1	2.5	Pass
	contains()	containsTestForLessThanLowerBound() Input: -13.1 Partition: B1	false	Pass

		containsTestForOnLowerBound() Input: -11.5 Partition: B1	true	Pass
		containsTestForInBetweenBounds() Input: 0 Partition: B1	true	Pass
		containsTestForOnUpperBound() Input: 31.5 Partition: B1	true	Pass
		containsTestForMoreThanUpperBound() Input: 41.5 Partition: B1	false	Pass
		containsTestMin() Input: Double.MIN_VALUE Partition: B1	true	Pass
		containsTestMax() Input: Double.MAX_VALUE Partition: B1	true	Pass
	equals()	equalsTestForSameRange() Input: range1: -10 to 10 range2: -10 to 10 Partition: A1 and A2	true	Pass
		equalsTestForLowerRange() Input: range1: -10 to 10 range2: -20 to 10 Partition: A1 and A2	false	Pass
		equalsTestForHigherRange() Input: range1: -10 to 10 range2: -10 to 20	false	Fail

		Partition: A1 and A2		

4 HOW THE TEAMWORK/EFFORT WAS DIVIDED AND MANAGED

Ryan Sommerville: Formatted and put together most of the report.

Eric Renno: Quality Control, Test design

Quinn Ledingham: Created test cases and implemented them in JUnit

Kaumil Patel: Created and implemented test cases.

5 DIFFICULTIES ENCOUNTERED, CHALLENGES OVERCOME, AND LESSONS LEARNED

In the beginning, we had some trouble with designing the tests. Initially coming up with the test plan caused some level of difficulty because we were not sure what exactly that consisted of. But after researching it a little it made more sense, and we knew where to begin. Designing the test cases was also at first a challenge because we did not know exactly what they should look like. After reading the slides again, however, we had a better idea of how to make them. JMock also caused a few problems just getting it set up and learning how to use it. Now we know how to get started designing a test plan and test cases and we know how to use JMock now.

6 COMMENTS/FEEDBACK ON THE LAB ITSELF

The instructions on how to set up the eclipse project were simple to follow and made it easy to get started working on the lab. The document was very well indexed, making it very easy to navigate over to different sections in the report using the table of contents. The steps described in the document were also very well put, with a lot of information given in regards to the testing tools. The requirements for this assignment were also clearly stated, leaving less room for confusion among group members. In conclusion, the lab report helped our group to gain a reasonable understanding of unit testing and how to plan and design tests.